A Tableau Based Automated Theorem Prover Using High Performance Computing

Md Zahidul Islam, Ahmed Shah Mashiyat†, Kashif Nizam Khan, and S.M. Masud Karim
Computer Science and Engineering Discipline, Khulna University, Khulna, Bangladesh
†Department of Mathematics, Statistics and Computer Science, St. Francis Xavier University, NS, Canada.
Email: zahid@cse.ku.ac.bd, †x2008ooc@stfx.ca, kashif570@yahoo.com, and masud@cse.ku.ac.bd

***Abstract*— Automated Theorem Proving systems are enormously powerful computer programs capable of solving immensely difficult problems. The extreme capabilities of these systems lie on some well-established proof systems, such as Semantic tableau. It is used to prove the validity of a formula by contradiction and it can produce a counterexample if it fails. It can also be used to prove whether a formula is a logical consequence of a set of formulas. Tableau can be used in propositional logic, predicate logic, modal logic, temporal logic, and in other non-classical logics. In this article, we describe a detailed implementation of a sequential tableau algorithm for propositional and first-order logic using a procedural language rather then logic programming language. We also illustrate a tableau based proof system in a distributed environment using the Message Passing Interface. This paper also investigates two distinct approaches for parallel and distributed implementation and describes the experimental formula generation procedure. The proposed high performance approach will un-wrap an efficient paradigm for automated theorem proving.**

***Index Terms*— Automated theorem proving, high performance computing, message passing interface, semantic tableau.**

I. Introduction

In 1994, the famous "Pentium bug" caused Intel a loss of $475M [1]. Since then, formal verification of hardware and software has gained popularity in industry. There are two main approaches to formal verification: model checking and theorem proving. Model checking explores a (finite) model and theorem proving explores logical derivations of a theory. Automated theorem proving is the process of proving mathematical theorems by a computer program in an automated fashion. Automated theorem proving is currently the most well developed sub-field of automated reasoning. The language in which the mathematical theorems or formulas are written is logic. As knowledge representation is often based on logic, many theorem provers have been successfully used to verify knowledge based systems [2]. More detail classification of verification approaches can be found in [3]. Among several theorem proving approaches, tableau is a widely used one. The main uses of the semantic tableaux method are validation check of a formula, to identify a logical consequence of a formula, and satisfiability checking of a set of formulas [4]–[6]. We discuss the tableau calculi for propositional logic, first-order logic and linear temporal logic to describe our proposed theorem proving approach. Massive search space generated during automated reasoning is considered as a critical problem in the field of theorem proving. Distribution and exploration of the search space in parallel is a motivating approach to handle this problem [7]. The aim of this paper is to investigate the scope of applying distributed memory computing to enhance the efficiency of theorem proving. This paper comprises the theoretical background, a description of the system architecture, details of the serial implementation and two different approaches to make it parallel. The proposed approach increases the memory capacity and the computing power by utilizing multiple processors of a cluster to manage the large search space.

The rest of the paper is structured as follows: section II introduces semantic tableaux, section III summarizes some existing tableau based theorem provers, section IV describes some implementation issues, section V describes two different parallel implementations, and a case study with the proposed implementation is discussed in section VI. Finally, section VII points out some related ongoing research and future directions and concludes the paper.

II. Semantic Tableaux

A proof procedure is an algorithm to prove the correctness of a logical formula. A semantic tableau is a complete and sound formal proof procedure which is specially suitable for automation [4]. Some characteristics of tableau procedures, as stated in [5], are: first, it is a refutation procedure, i.e., to prove a formula ϕ valid we try to prove its negation $\neg\phi$ satisfiable. Next, the expression ϕ is broken down into several subformulas by applying expansion rules. Finally the construction procedure closes, based on its syntax. If each case closes, the tableau is said to be closed. A closed tableau beginning with $\neg\phi$ is a tableau proof of ϕ. In a tableau, the formulas obtained by applying expansion rules are always simpler than the formulas they come from. Tableau is well-suited for the discovery of proofs, either by hand or by machines. Once quantifiers are added, complications may arise. However, there are various methods to handle those quantifiers [4], [5]. For this paper, we restrict our discussion to the tableau for propositional logic.

doi:10.4304/jcp.7.3.597-607

The tableau method can be applied on both signed and unsigned formulas. Here we consider only the signed formulas. Signed formulas can be defined as, "*A signed formula $T\phi$ is called true if ϕ is true and false if ϕ is false. And a signed formula $F\phi$ is called true if ϕ is false and false if ϕ is true*". The tableau method is based on the following eight facts. These facts hold under any interpretation of any formulas ϕ and ψ: (1) If $\neg\phi$ is true, then ϕ is false, (2) If $\neg\phi$ is false, then ϕ is true, (3) If a conjunction $\phi \wedge \psi$ is true, then ϕ, ψ are both true, (4) If a conjunction $\phi \wedge \psi$ is false, then either ϕ is false or ψ is false, (5) If a disjunction $\phi \vee \psi$ is true, then either ϕ is true or ψ is true, (6) If a disjunction $\phi \vee \psi$ is false, then ϕ, ψ are both false, (7) If $\phi \rightarrow \psi$ is true, then either ϕ is false or ψ is true, and (8) If $\phi \rightarrow \psi$ is false, then ϕ is true and ψ is false.

From the above facts we observe, there are two types of expansions - deterministic and nondeterministic. In a deterministic expansion, there is exactly one possible consequence (facts 1, 2, 3, 6 and 8). In a nondeterministic expansion, there is more than one possible consequence (facts 4, 5 and 7).

A. Tableaux Construction Rules

Based on the eight facts stated in earlier the tableau expansion rules, as in [6], are shown below in Figure 1. For each logical connective ($\neg$, $\wedge$, $\vee$, $\rightarrow$), there are two rules: one preceded by the sign T and the other by F.

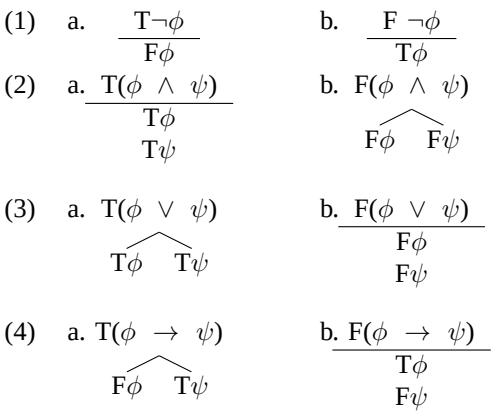

Figure 1. Tableaux expansion rules for propositional logic

We can categorize these rules similarly as [6], category (A) those having direct consequences ($T\neg\phi$, $F\neg\phi$, $T(\phi \wedge \psi)$, $F(\phi \vee \psi)$ and $F(\phi \rightarrow \psi)$) and category (B) those having branches ($F(\phi \wedge \psi)$, $T(\phi \vee \psi)$ and $T(\phi \rightarrow \psi)$).

While constructing a tableau, if a formula of type (A) occurs, we simultaneously subjoin its consequences into all branches below it. Similarly, when a formula of type (B) occurs, we divide all branches that pass through the line into sub-branches. Once we apply a rule to any formula, that formula is never expanded again.

B. Illustration of the Tableaux Method

Now let us see an example of tableau construction by applying the rules discussed so far. Let us consider, we wish to prove the formula $\phi = [p \vee (q \wedge r)] \rightarrow [(p \vee q) \wedge (q \vee r)]$, where p, q and r are atomic propositions. The complete tableau is given in Figure 2; explanation follows immediately. The number to the left of a line is used for identification purpose only; they are not needed in actual construction.

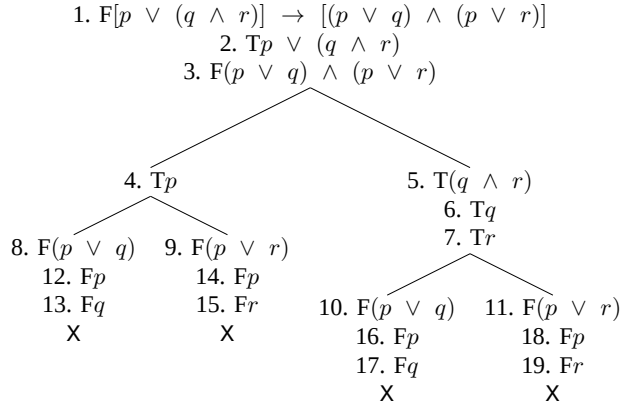

Figure 2. Tableau for the formula $\phi = [p \vee (q \wedge r)] \rightarrow [(p \vee q) \wedge (p \vee r)]$

To prove the formula, we need to drive a contradiction from the initial assumption that it is false. So we will start with the formula preceded by F. Now the formula is of the form $F(\phi \rightarrow \psi)$ which is false only when ϕ is true and ψ is false (fact 8, rule 4(b)). We write (2) and (3) as a direct consequence of (1). Now (2) is of the form $T(\phi \vee \psi)$, according to fact 5 (rule 3(a)) we cannot draw any direct conclusion from (2). Similarly, we cannot draw any conclusion from (3) (fact 4, rule 2(b)). So from (2) the tableau branches to two possibilities (4)$T\phi$ and (5)$T\psi$. Line (5) i.e., $T(q \wedge r)$ immediately produces Tq and Tr. Similarly when we explore all the formulas, we obtain the tableau tree in Figure 2.

Now at the left most branch (12) is a direct contradiction of (4). So we got a contradiction on this branch, and we can close this branch.

Similarly (14) contradicts (4), (17) contradicts (6), and (19) contradicts (7). Thus all branches are closed, i.e., they lead to a contradiction. Thus the formula $\phi = [p \vee (q \wedge r)] \rightarrow [(p \vee q) \wedge (q \vee r)]$ can never be false in any interpretation, so it is valid. The symbol 'X' at the end of each branch indicates the corresponding branch is closed; we can close a branch as soon as a contradiction is found.

C. Tableau for First-Order Logic

We now describe the tableau method for first-order logic. As with propositional logic, proofs will be binary branching trees, whose nodes are now labeled by first-order formulas. In first-order tableau construction we will add four additional rules for the two quantifiers. We have four quantifier rules - one for each of $T\forall\, x\phi(x)$, $F\forall\, x\phi(x)$, $T\exists\, x\phi(x)$, $F\exists\, x\phi(x)$. One point to be noted that, none of these rules are branching.

Rule $T\forall$: From $T\forall\ x\phi(x)$, we may directly infer $T\phi(a)$, where a is any parameter. **Rule** $T\exists$: From $T\exists\ x\phi(x)$, we may directly infer $T\phi(a)$, where a is a new parameter i.e., has not been used yet. **Rule** $F\forall$: From $F\forall\ x\phi(x)$, we may directly infer $F\phi(a)$, where a is a new parameter as in **Rule** $T\exists$. **Rule** $F\exists$: From $F\exists\ x\phi(x)$, we may directly infer $F\phi(a)$, where a is any parameter.

These four rules are shown in Figure 3.

Rule $T\forall$: $\dfrac{T\forall\ x\phi(x)}{T\phi(a)}$ (a is any parameter)

Rule $F\exists$: $\dfrac{F\exists\ x\phi(x)}{F\phi(a)}$ (a is any parameter)

Rule $T\exists$: $\dfrac{T\exists\ x\phi(x)}{T\phi(a)}$ (a is a new parameter)

Rule $F\forall$: $\dfrac{F\forall\ x\phi(x)}{F\phi(a)}$ (a is a new parameter)

Figure 3. Tableaux expansion rules for first-order logic

Rules $T\forall$ and $F\exists$ are collectively called $universal$ rules. Similarly **Rules** $T\exists$ and $F\forall$ are collectively called $existential$ rules. Decidability is a limiting factor in use of tableau method for first-order logic. For invalid formulas the tableau for first-order logic may lead to an infinite tableau. In fact, there is no correct and complete proof method for first-order logic that always terminates, as first-order logic is undecidable by definition. However in some cases (specifically when a branch represents a Hintkka set), the tableau method can decide whether a formula is invalid in the middle of the proof procedure.

D. Tableau for Linear Temporal Logic (LTL)

Efficient tableau-based methods for satisfiability testing and model checking of temporal formulae have been developed since the early 1980s: for the linear-time logic LTL (by Wolper [8]); for the branching-time logic UB (By Ben-Ari et al. [9]) and CTL (Emerson et al. [10]). Unlike automata-based approach for model checking and satisfiability checking of temporal logics, tableau-based methods are less developed and applied for industry applications, but are more natural and intuitive from logical perspective, easier for execution by humans, and (arguably) potentially more flexible and practically efficient, if suitably optimized [11]. Tableau based methods are one of the main methods for automating deduction in many temporal logics like LTL and CTL [12]. In this section we outline tableau based decision procedures for satisfiability of LTL formulae. For convenience of the reader not previously familiar with tableau-based decision procedures for LTL, we start our discussion with some preliminary concepts.

1) Syntax and semantics of LTL: LTL models time implicitly in along a path in the future. The syntax of LTL includes some temporal operators along with the propositional operators. The inductive definition of LTL in Backus Naur Form (BNF) is given below:

$$\begin{aligned}\phi\ :=\ &\top\mid\bot\mid p\mid\neg\phi\mid(\phi\ \wedge\ \psi)\mid(\phi\ \vee\ \psi)\mid\\ &(\phi\ \rightarrow\ \psi)\mid X\phi\mid F\phi\mid G\phi\mid\phi\ U\ \psi\end{aligned}\qquad(1)$$

where p ranges over a finite set of proposition.

Many different semantics of LTL are available in literature. We adopt the semantics discussed in [13]. In this framework LTL formulae are interpreted over a Kripke structure. Let $\mathcal{M}=(S,\rightarrow,L(S))$ be such a structure and ϕ is an LTL formula. $\pi=s_1\rightarrow\cdots$ be a path in $\mathcal{M}$. The semantics of LTL is defined as follows:

1) $\pi\models\top$
2) $\pi\not\models\bot$
3) $\pi\models p$ iff $p\in L(s_1)$
4) $\pi\models\neg\phi$ iff $\pi\not\models\phi$
5) $\pi\models(\phi\wedge\psi)$ iff $\pi\models\phi$ and $\pi\models\psi$
6) $\pi\models(\phi\vee\psi)$ iff $\pi\models\phi$ or $\pi\models\psi$
7) $\pi\models(\phi\rightarrow\psi)$ iff $\pi\models\psi$ whenever $\pi\models\phi$
8) $\pi\models X\phi$ iff $\pi^2\models\phi$
9) $\pi\models G\phi$ iff $\forall i,i\geq 1,\pi^i\models\phi$
10) $\pi\models F\phi$ iff $\exists i,i\geq 1$ such that $\pi^i\models\phi$
11) $\pi\models\phi U\psi$ iff $\exists i,i\geq 1$ such that $\pi^i\models\psi$ and $\forall j,j=1,\cdots,i-1$ we have $\pi^j\models\phi$

We write π^i for the path starting at s_i.

2) Tableau based decision procedure for LTL: Tableau for LTL is different than those for propositional and first-order logic. The complications in LTL tableau arise due to the fixpoint operators, as they require a special mechanism for checking realization of eventualities (the U operator). In LTL, Boolean connectives are handled similarly as for propositional logic and temporal connectives are handled by decomposing them into a requirement on the "current state" and a requirement on "the rest of the sequence"'. As discussed in [11], [12], [14], tableau for LTL requires the notion of Hintikka structures. Generally tableau for LTL involves two phases. In the first phase a tableau tree is constructed by applying tableau construction rules on the initial formula. Then in the second phase, the inconsistent nodes in the tree are removed from the tree. Inconsistent nodes are those not satisfying in any Hintikka structure. In LTL there are two types of inconsistencies - one is the propositional inconsistency (p and $\neg p$ in the same branch of the tableau tree) and the other one is any unfulfilled eventuality in a node. If the root node is removed during the second phase the initial formula is not satisfiable. Otherwise there is a Hintikka structure in the tableau tree satisfying the initial formula and therefore the initial formula is satisfiable. A convenient implementation direction of the tableau for LTL is available in [15].

III. EXISTING TABLEAU THEOREM PROVERS

In this section, we summarize some of the existing tableau theorem provers found in various literatures with an intention of providing a brief overview of the logic and the implementation language used.

A. ${}_3T^AP$

${}_3T^AP$ [16] has been developed at the University of Karlsruhe. It can deal with two-valued-first-order predicate logic as well as any finite-valued first-order logic. It can also deal with equality, which is treated as a two-valued predicate in the multiple-valued case. ${}_3T^AP$ is a sequential theorem prover implemented using Prolog.

B. Meteor

Meteor [17] compiles clauses into a data structure which is inferred by a sequential or parallel inference engine. The underlying inference mechanism of Meteor is model elimination with additional redundancy reducing mechanisms. It can prove first-order predicate logic in clausal form and developed using unix based C programming language.

C. Parthenon

Parthenon (PARallel THEorem prover for NON-Horn clauses) [18] is a parallel theorem prover for first-order predicate logic. The underlying proof calculus is a variant of model elimination. Parthenon is the first implementation of an OR-parallel theorem prover, which runs on various multiprocessors capitalizing C-threading. It is implemented using a computational model similar to the SRI-Model for OR-parallel Prolog.

D. PTTP

The Prolog Technology Theorem Prover (PTTP) [19] is an implementation of the model elimination theorem-proving procedure. Instead of Prolog's unbounded depth first search, it uses the depth-first iterative-deepening search to make the search strategy complete. It can prove theorems written in first-order predicate logic. This sequential theorem prover is implemented using Prolog and Lisp (Formally known as LISP).

E. SETHEO

The SEquential THEOrem prover SETHEO [20] is also based on the model elimination calculus. The tool ensures the completeness by iterative deepening using several depth measures. Based upon the efficient handling of syntactic un-equality constraints, a number of mechanisms for pruning the search space have been used in this tool. SETHEO is implemented using C and both sequential and parallel versions of SETHEO are available. Several parallel provers such as PARTHEO, SPTHEO and RCTHEO are developed on top of SETHEO.

F. LoTREC

LoTREC is a generic tableau theorem prover for modal logic. It is a suitable educational tool for students and researchers for creating, testing and analyzing tableau method implementations. LoTREC is sequential and developed using Java [21] with an aim of flexibility, portability, and nice interface.

G. MLTP

MLTP [22] implements a labeled tableau calculus, aiming to maximize both efficiency and generality. Several optimization techniques for modal logic reasoning are incorporated in MLTP. To facilitate generic reasoning, MLTP incorporates a generic data structure, a low-level macro language, an event-handler framework and a high-level specification language. MLTP is sequential and developed using C++.

H. FaCT++

Tableau method is also used as a reasoner for higher order logic. FaCT++ [23] is a tableaux-based reasoner for OWL Description Logics (DLs), implemented in C++. It can be used as a standalone DIG reasoner and as a back-end reasoner for the OWL API-based applications.

I. Pellet

Pellet [24] is an OWL DL reasoner based on the tableaux algorithms developed for Description Logics. Pellet is open-source and developed using Java. Pellet can be used with OWL API libraries and it provides a DIG interface.

IV. Implemantation of the theorem prover

In this section, we discuss the issues related to sequential implementation of the theorem prover including the syntax of formula representation, data structure, expansion rule representation, terminating conditions, and the tableau construction algorithm. We also discuss two different approaches to make it parallel.

A. Syntax

The proposed algorithm takes the input represented by a string, and returns the tableau tree in *inorder* with the closed branches marked 'closed'. The implementation supports all the usual Boolean connectives represented as: *and* for $\wedge$, *or* for $\vee$, *implies* for $\rightarrow$ and *not* for $\neg$. The formulas can be defined in BNF as:

$$\phi \ := \ p \mid not\ \phi \mid (\phi\ and\ \psi) \mid (\phi\ or\ \psi) \mid (\phi\ implies\ \psi)$$

Where p stands for any atomic proposition and each occurrence of ϕ to the right of $:=$ stands for any already constructed formula.

B. Data Structures

1) Formula Representation: In our implementation, a formula is stored in a tree structure as in [13], [22]. Advantages of the tree structure include, easy extracting sub-formulas, which is, in fact, the most common operation during derivations; easy to verify that a formula is well formed; and obviously in a tree representation we do not need to search for an operator to apply an expansion rule, as it is the root of the formula tree. The tree representation of the formula $(p\ or\ (q\ and\ r))\ implies\ ((p\ or\ q)\ and\ (p\ or\ r))$ and the sub formulas after applying the expansion rule are depicted in Figure 4 and Figure 5.

2) Tableau Tree Representation: The tableau is a tree, made up of nodes. Every node of the tree contains the following fields.

`leftNode:` A pointer to the left subtree of the current node.
`rightNode:` A pointer to the right subtree of the current node.

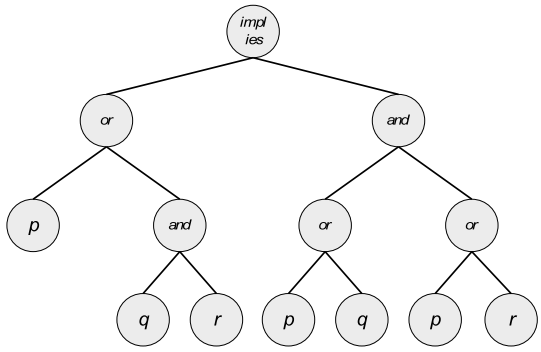

Figure 4. Tree representation of the formula $(p\ or\ (q\ and\ r))\ implies\ ((p\ or\ q)\ and\ (p\ or\ r))$

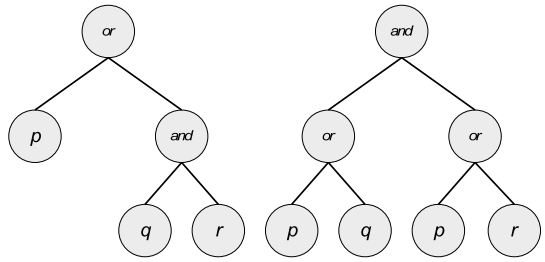

Figure 5. The two sub formulas after applying expansion rule

`parentNode`: A pointer to the parent of the current node.
`formulaList`: A pointer to the list of formulae of the current node.
`closed`: A boolean flag indicating whether the corresponding branch is closed or not.

The `formulaList` pointer in a tableau node points to a list where each node in the list is a formula with some additional information as follows.

`index`: A unique integer identifier for the node.
`sign`: A boolean flag used to indicate the sign of the current formula.
`formula`: A pointer to the formula tree of the current node.
`status`: A boolean flag indicating whether the formula is processed or not.

Besides the two main data structures described above, some conventional data structures like list, stack and queue are used to manipulate the formula and tableau nodes.

C. Expansion Rules Representation

For simplicity, we used a unifying scheme as [4], [6], in our implementation. This scheme classifies formulae into two groups based on their logical connectives. Conjunctive type formulas are called α-formula and disjunctive type formulas are called β-formula. α and β-formulas are shown in Table I below.

TABLE I.
α AND β-FORMULAS

α	α_1	α_2
$T(\phi\ \wedge\ \psi)$	$T\phi$	$T\psi$
$F(\phi\ \vee\ \psi)$	$F\phi$	$F\psi$
$F(\phi\ \rightarrow\ \psi)$	$T\phi$	$F\psi$
$T(\neg\ \phi)$	$F\phi$	$F\phi$
$F(\neg\ \phi)$	$T\phi$	$T\phi$

β	β_1	β_2
$T(\phi\ \vee\ \psi)$	$T\phi$	$T\psi$
$F(\phi\ \wedge\ \psi)$	$F\phi$	$F\psi$
$T(\phi\ \rightarrow\ \psi)$	$F\phi$	$T\psi$
$T(\ \neg\ \phi)$	$F\phi$	$F\phi$
$F(\ \neg\ \phi)$	$T\phi$	$T\phi$

Using these α, β notations now we need only two expansion rules: if an α-formula occurs on a branch replaces it with α_1 followed by α_2 on same branch and if a β-formula occurs on a branch split the branch in two, one with β_1 and other one with β_2.

D. Terminating the Tableau Construction

It is very important to identify when to terminate the construction of the tableau tree. In our implementation, a tableau construction terminates in one of two cases: when all the branches are closed or when there is no more formula to process. A branch in the tableau tree closes as soon as a contradiction appears on that branch. A contradiction is of the form of a formula with both *true* and *false* sign ($F\phi, T\phi$) on the same branch. The `status` of all the formulas is *true* (section IV.B.2) indicates there is no more formula to process. The `closed` is marked *true* (section IV.B.2) only if a contradiction is found.

E. The Tableau Algorithm

At the beginning, a *tableau tree* is constructed from the input formula. We define the root of the tableau tree with attributes `leftNode`, `rightNode`, `parentNode`, `formulaList`, `closed`. Initially, this tableau root node is added to a queue `tableauNodeQueue`. The algorithm skeleton is provided in Algorithm 1.

When the expansion rule is applied to a formula the value of its `status` attribute is assigned to *true*. The new formulas created as a expansion of the formula is assigned `false` to their status. The main steps of the algorithm continues until the `tableauNodeQueue` is empty or all the branches are closed. The empty `tableauNodeQueue` represents that all formulas are processed. Checking for closed tableau involves two types of search: First, whenever a α-formula is processed, the contradiction of sub-formulas α_1 is searched first in the local `formulaList` of the current node then the `formulaList` of its parent is searched until the root is reached. If a contradiction is found the `closed` flag of the node containing the sub-formula is set *true*. Sub-formula α_2 is searched in the same way. A β-formula is handled in a different way. A β-formula creates two new `tableauNode` with two sub-formulas (β_1 and β_2) in their `formulaList`, so we do not need to search the local `formulaList`, starting at the `formulaList` of the parent up to the `formulaList` of the root is searched for contradiction. The second type of search

Algorithm 1: TableauAlgorithm

Data: Input formula.
Result: True or Counter Example
begin
  Instantiate `formulaList` node with attributes:
    `index` $\longleftarrow 0$
    `sign` $\longleftarrow false$
    `formula` $\longleftarrow$a pointer to the input formula tree
    `status` $\longleftarrow false$
  Instantiate the root node with attributes:
    `leftNode` $\longleftarrow null$
    `rightNode` $\longleftarrow null$
    `parentNode` $\longleftarrow null$
    `formulaList` $\longleftarrow$a pointer to the formulaList
    `closed` $\longleftarrow false$
  Add root node to `tableauNodeQueue`
  while *`tableauNodeQueue`* $\neq \emptyset \vee$ *all the branches are closed* **do**
    `currentNode` $\longleftarrow$ $front(tableauNodeQueue)$
    for *all formulas* $\in$ *`formulaList` of `currentNode`* **do**
      if α*-formula* **then**
        Expand to two new sub formulas with status $false$
        Add the new sub formulas to the end of `formulaList`
      else
        add $\beta - formula$ to `betaFormulaQueue`
    for *formulas* $\in$ *`betaFormulaQueue`* **do**
      if β*-formula* **then**
        Expand the formula
        create two new tableau nodes with
        `parentNode` $\longleftarrow$ `currentNode`
        Two formulas are added to `formulaList` of newly created nodes

occurs whenever a branch closes. The tableau construction terminates when all the branches closes. To check this condition we need to search the whole tableau tree. To find contradiction, when a branch closes, the tableau tree starting at the root is searched in inorder.

F. The Tableau Algorithm for LTL

This section gives an informal algorithm for LTL tableau based on the discussion in section II-D.2. Our algorithm is analogous to the apporaches in [9], [15], [25]. The algorithm first constructs the tableau tree applying construction rules and then removes the inconsistent nodes. The construction rules are defiend using Smullyan's α-, β-notation given in Figure 6.

The construction starts with an LTL formula φ as the root. We use U_{n_i} to represent the set of formulas at a node n_i. Let n_0 be the root of the tableau tree $\mathcal{T}_\varphi$, the construction begins with $U_{n_0} = \{\varphi\}$. Following rules are then applied to construct the final tableau tree.

α rule:If $\alpha \in U_n$, then create a single child n_1 of n with $U_{n_1} = U_n \cup \{\alpha_1, \alpha_2\}$

β rule:If $\beta \in U_n$, then create two children n_l and n_2 of n with $U_{n_1} = U_n \cup \{\beta_1\}$ and $U_{n_2} = U_n \cup \{\beta_2\}$

When any more application of α, β rules is not possible, the following X rule is applied.

X rule:If $X\varphi \in U_n$, then create a single child n_1 of n with $U_{n_1} = U_n \cup \{\varphi\}$

Applications of α-, β-rules (except for operators $\wedge, \vee$) may create a node with same formulas. This situation is handled using a "feedback" edge instead of creating a new node. The construction of $\mathcal{T}_\varphi$ terminates by the following two rules.

T1: If $\{\varphi, \neg\varphi\} \in n$, (this is same as the propositional tableau) this node is closed and is not expanded further.

T2: If a node m is about to be created as a child of a node n and there is an ancestor n' of n such that $U'_n = U_m$, then without creating m, n and n' are connected with a "feedback" edge.

After the construction we apply the following rules to eliminate the unsatisfiable nodes of $\mathcal{T}_\varphi$.

E1: If $\{\varphi, \neg\varphi\} \in n$, then eliminate n.

E2: If all the successor of n have been eliminated, then eliminate n.

E3: If $X\varphi$ or $\varphi\ U\ \varphi' \in n$ and
If there is no path in $\mathcal{T}_\varphi$ from n to n' such that $\varphi' \in n'$, then eliminate n.

The decision procedure terminates after all unsatisfiable nodes have been eliminated. If the root n_0 has been eliminated, then φ is not satisfiable otherwise it is satisfiable.

V. PARALLEL TABLEAU THEOREM PROVER

The problem of deciding the validity of a formula varies from trivial to impossible, depending on the underlying logic. For the frequent case of propositional logic, the problem is decidable but NP-complete, and hence only exponential-time algorithms are believed to exist for general proof procedures. This time consuming computational tasks need to be addressed by high performance computing power, by employing the aggregate power of network-interconnected clusters; then dividing the computation task between multiple processors using message passing interface (MPI). Identifying parallelism opportunities can be a difficult task. We are parallelizing the theorem prover by splitting the formula. We found two parallel architectures (described later) that fit our purpose. We have chosen Java as the implementation language and specifically we are using mpiJava [26] to

α	α_1	α_2
$\neg\neg\phi$	ϕ	ϕ
$\phi \wedge \psi$	ϕ	ψ
$\neg(\phi \vee \psi)$	$\neg\phi$	$\neg\psi$
$\neg X\phi)$	$X\neg\phi$	$X\neg\phi$
$G\phi$	ϕ	$XG\phi$

β	β_1	β_2
$\neg(\phi \wedge \psi)$	$\neg\phi$	$\neg\psi$
$\phi \vee \psi$	ϕ	ψ
$\phi\ U\ \psi$	ψ	$\phi \wedge X(\phi\ U\ \psi)$
$\neg(\phi\ U\ \psi)$	$\neg\psi \wedge \neg\phi$	$\neg\psi \wedge \neg X(\phi\ U\ \psi)$
$\neg G\phi)$	$\neg\phi$	$\neg XG\phi$

Figure 6. α and β-formulas for LTL.

implement the parallel version of the theorem prover. The intended architectures are discussed below. The following architectures are based on the Single Program Multiple Data (SPMD) model. Both the approaches are inspired by [7], [27].

A. Distributed Memory Computing

A distributed memory algorithm runs on many individual computers. Each computer processes a small piece of input data with its own computation resources. Communication between participating computers is needed to complete the computation for exchanging input data and computed results. There are several network primitives used by distributed processes to communicate with each other. The basic primitives are *send* and *receive*. A *send* message function provides a process the possibility to send a content of a block of memory to another process and allows the sender to put in a message type identifier. The recipient of the message is specified by its logic identification rather than a physical address of the workstation. A *receive* function calls a block of memory with the content of a received message and supplies the receiving process with the enclosed tags to allow proper interpretation of the received data.

A computation of a distributed memory algorithm should terminate when the algorithm has reached a situation that no process can progress further in the computation. We can ensure this by providing the fact that there is no message in transit. Unfortunately, to learn this, the processes need to communicate, i.e. to send and receive messages. This problem is generally referred to as a distributed termination detection problem in literature. For this reason we used two parallelizing technique to implement our parallel tableau algorithm. The first approach is a tree based approach where processes on the same branch communicate to detect termination. The second approach uses a centralized work pool where the master process keeps track of the terminated processes and each process notifies the master when it terminates.

B. Tree Based Partitioning Approach

In this approach, the tree structure of the tableau is preserved. The root tableau node is assigned to the processor p0. Whenever a new node is created by processing a β-formula the left child node is sent to the processor $2 \times id + 1$ and right child is sent to $2 \times id + 2$, where id is the rank of parent processor as shown in Figure 7. The child processors can now work concurrently. The algorithm skeleton is provided in Algorithm 2.

Algorithm 2: TreeBasedPartitioning

```
procedure Workstation( formula, i )
begin
    if formula ∈ β − formula then
        leftFormula, rightFormula ⟵
        Expand(β − formula)
        Workstation( leftFormula, 2 × i + 1 )
        Workstation( rightFormula, 2 × i + 2 )
    else
        Instantiate formulaList node
        Instantiate the root node and add to
        tableauNodeQueue
        while tableauNodeQueue ≠ ∅ ∨ all the
        branches are closed do
            currentNode ⟵
            front(tableauNodeQueue)
            for all formulas ∈ formulaList of
            currentNode do
                if α-formula then
                    Expand to two new sub formulas
                    with status false
                    Add the new sub formulas to the
                    end of formulaList
                else
                    add β − formula to
                    betaFormulaQueue
```

In case of searching, a processor can calculate its parent's id using the formula $floor((id-1) \div 2)$, where id is the rank of the child processor as depicted in Figure 8. The tree structure of the tableau construction made this approach an automatic choice.

C. Centralized Work Pool Approach

There are some efficiency issues in the tree based approach, as for example if work load is not distributed equally, some processors may sit idle when others are working. To overcome these problems, we can adopt centralized work pool approach. In this approach the tree structure of the tableau tree is not preserved, the nodes are considered to have only one formula list. When a β-formula is processed two new nodes are created with two subfomulas and the unprocessed β-formula along with the formulas in the current node is copied to the formula list of the new nodes. New nodes are then added to the dynamic work pool. So to search for a contradiction

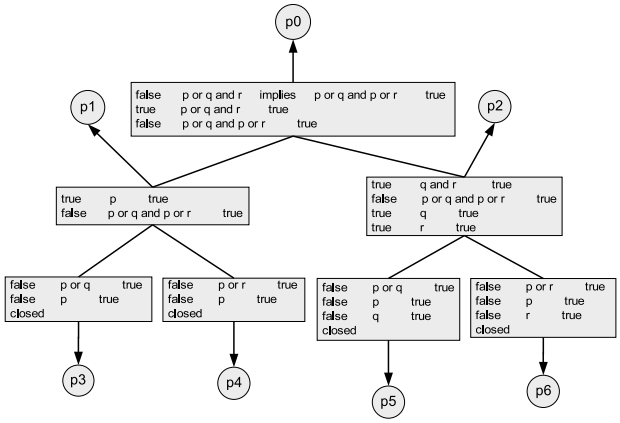

Figure 7. Parallel distribution of the tableau tree

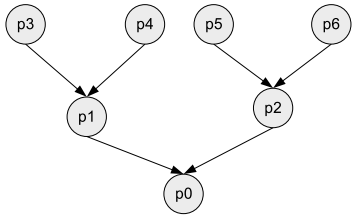

Figure 8. Communication between the processors in searching

involves searching the local formula list only. A task queue is maintained by the master processor. Each slave processor only requests a new task when it is idle. The master sends the slave a task from the queue if the `processed` flag of the node is *false*. So a termination can be detected when all the slaves are idle and the processed flag of all the nodes in the task queue is *true*. To find out whether the input formula is satisfiable, the `closed` flag of the nodes in the task queue are checked. If the `closed` flag of all the nodes are *true*, we get a closed tableau, i.e., the input formula is satisfiable otherwise the formula is not satisfiable. A schematic view of the work pool approach is shown in Figure 9.

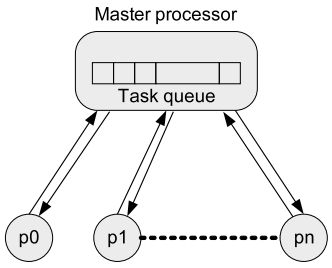

Figure 9. Centralized work pool distribution of the tableau tree

VI. CASE STUDY

In this section we describe our experimental results. We show a representation of reasoning about classical *n*-queen problem. The problem is to place *n*-queens on a $n \times n$ chessboard so that no two queens dominate each other. In order to specify this problem we introduce n^2 propositional variables; each associated with one square of the chess board. For convenience we show a toy example with only *4*-queens. Figure 10(a) shows the solution to the problem; 10(b) shows an illegal configuration and 10(c) shows a legal configuration. Our implementation can conclude that (b) is illegal and (c) is legal by applying the tableau proof system discussed earlier.

A chess board is a 2D grid, where [1,2] represents the square at row 1 and column 2. Let us assume that the symbol $q_{1,1}$ represents "There is a queen in [1,1]", then $\neg q_{1,1}$ means "There is no queen in [1,1]". The rules of *4*-queen can be represented as follows:

Rule 1: At least one queen in row 1.

$$q_{1,1} \vee q_{1,2} \vee q_{1,3} \vee q_{1,4}$$

Similarly ***Rule 2***, ***Rule 3*** and ***Rule 4*** are defined for row 2, 3 and 4.

Rule 5: At most one queen in row 1.

$$\begin{aligned}
q_{1,1} &\rightarrow \neg q_{1,2} \wedge \neg q_{1,3} \wedge \neg q_{1,4} \\
q_{1,2} &\rightarrow \neg q_{1,1} \wedge \neg q_{1,3} \wedge \neg q_{1,4} \\
q_{1,3} &\rightarrow \neg q_{1,1} \wedge \neg q_{1,2} \wedge \neg q_{1,4} \\
q_{1,4} &\rightarrow \neg q_{1,1} \wedge \neg q_{1,2} \wedge \neg q_{1,3}
\end{aligned}$$

Rule 6, ***Rule 7*** and ***Rule 8*** for row 2, 3 and 4 are defined similarly.

Rule 9: At least one queen in column 1.

$$q_{1,1} \vee q_{2,1} \vee q_{3,1} \vee q_{4,1}$$

Similarly ***Rule 10***, ***Rule 11*** and ***Rule 12*** are defined for column 2, 3 and 4.

Rule 13: At most one queen in column 1.

$$\begin{aligned}
q_{1,1} &\rightarrow \neg q_{2,1} \wedge \neg q_{3,1} \wedge \neg q_{4,1} \\
q_{2,1} &\rightarrow \neg q_{1,1} \wedge \neg q_{3,1} \wedge \neg q_{4,1} \\
q_{3,1} &\rightarrow \neg q_{1,1} \wedge \neg q_{2,1} \wedge \neg q_{4,1} \\
q_{4,1} &\rightarrow \neg q_{1,1} \wedge \neg q_{2,1} \wedge \neg q_{3,1}
\end{aligned}$$

Rule 13, ***Rule 14*** and ***Rule 15*** for column 2, 3 and 4 are defined similarly. For diagonal positions the rules are defined as follows: No two queen can be diagonally dominating each other.

Rule 16: For a queen in [1,1].

$$q_{1,1} \rightarrow \neg q_{2,2} \wedge \neg q_{3,3} \wedge \neg q_{4,4}$$

Rule 17: For a queen in [1,2].

$$q_{1,2} \rightarrow \neg q_{2,1} \wedge \neg q_{2,3} \wedge \neg q_{3,4}$$

Rule 18: For a queen in [1,3].

$$q_{1,3} \rightarrow \neg q_{2,2} \wedge \neg q_{3,1} \wedge \neg q_{2,4}$$

Rule 19: For a queen in [1,4].

$$q_{1,4} \rightarrow \neg q_{2,3} \wedge \neg q_{3,2} \wedge \neg q_{4,1}$$

And so on... for other positions.

To verify whether we can put a queen in [2, 1] when there

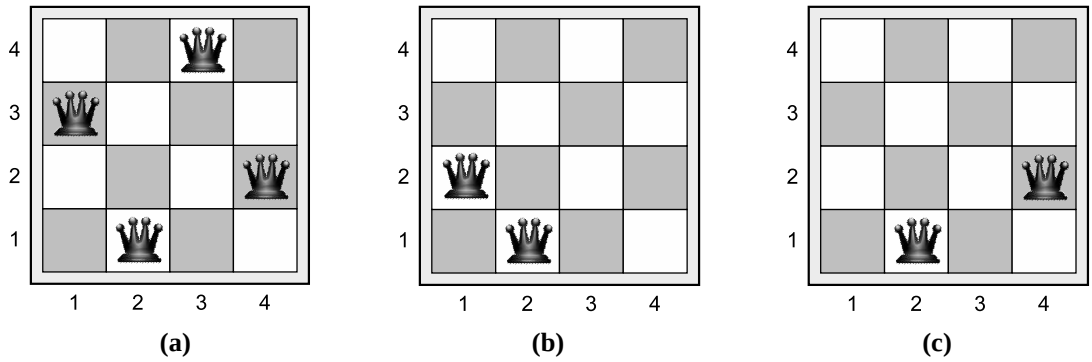

Figure 10. The *4*-queen problem: (a) an actual solution; (b) an illegal configuration and (c) a legal configuration.

is a queen in [1, 2], the input is given in the following form:

$$\phi := F(KB \ \wedge \ q_{1,2} \ \wedge \ q_{2,1})$$

If this returns a closed tableau then we can conclude $q_{2,1}$ is consistent with the KB i.e., a legal move. This representation using propositional logic is too tedius. Using first order logic the knowledgebase can be represented in a concise way. To represent the *4*-queen problem using first-order logic, we made the following assumptions.
Predicates: $Queen(x, y)$ represents a queen in [x, y].
Constants: 1, 2, 3, 4 represents row and column numbers.
Row constraints:
(i) At least one queen in each row.

$$\forall_i \ \exists_j \ Queen(i, j)$$

(ii) At most one queen in each row.

$$\forall_{i,j_1,j_2} \ Queen(i, j_1) \ \wedge \ j_1 \ \neq \ j_2 \ \rightarrow \ \neg \ Queen(i, j_2)$$

Column constraints:
(i) At least one queen in each column.

$$\forall_i \ \exists_j \ Queen(j, i)$$

(ii) At most one queen in each column.

$$\forall_{i,j_1,j_2} \ Queen(j_1, i) \ \wedge \ j_1 \ \neq \ j_2 \ \rightarrow \ \neg \ Queen(j_2, i)$$

Diagonal constraints:
At most one queen in [1,1], [2,2], [3,3] and [4, 4] .
$\forall_{i_1,j_1,i_2,j_2} \ Queen(i_1, j_1) \ \wedge \ i_1 \ = \ j_1$
$\rightarrow \ \neg \ Queen(i_2, j_2) \wedge \ i_2 \ = \ j_2 \ \wedge \ i_1 \ \neq \ i_2$
Similarly other constraints for all diagonal places can be defined. In this paper we show the tableau implementation for propositional logic only and we are wroking on it's extension to first order logic. The tableau for first order logic is discussed in section II-C. We have tested our implementation with 8-queen, 9-queen and 10-queen problems. However due to lack of available resources we could not compare our results with other theorem provers. We wish to compare our results with others in future.

VII. Conclusion

This paper describes the requirements, issues, process, and implementation to develop a distributed tableau theorem prover. Though our discussion is limited to propositional, first-order, and LTL tableau only, this approach can also be extended to other higher order logic. This type of parallel theorem prover is still not available, some earlier efforts can be found in [7], [27]. Though Prolog is a natural choice for the implementation of tableau algorithms, there are several difficulties in programming in Prolog. The highly unstructured nature of logic programs made it difficult to reuse the code. Extensive training is required for some new comer to understand the inner workings of a logic program written in Prolog. One of the key research problems in developing an automated theorem prover is to provide an easy to use interface to the user. Java suits well to provide the users with an easy to use interface to interact with the theorem prover, hiding the tedious details of the proof procedure. Considering these difficulties, we provide a framework that can be used to develop a tableau based algorithms using Java (or any high level programming language). In our previous work [28], we described the sequential implementation of the tableau theorem prover for only propositional logic and outlined two parallel approaches to make the theorem prover parallel. We have successfully developed the parallel versions of the proposed theorem prover and perform some experiments. In this paper, we provide a detailed description of building the test formulas which is used by the parallel theorem prover. Due to the tedious process of generating test formulas, we could not have sufficient formulas to provide a performance comparison of these two methods. In future, we wish to measure the performance of the two and identify the better one.

Acknowledgment

The authors wish to thank Kazi Shah Newaz Ripon for numerous discussions concerning this work, Abu Shamim Mohammad Arif and Kamrul Hasan Talukder for their assistance with implementation, and the anonymous reviewers of ICCIT 2010 for their valuable comments.

References

[1] T. Coe, T. Mathisen, C. Moler, and V. Pratt, "Computational aspects of the pentium affair," *IEEE Comput. Sci. Eng.*, vol. 2, pp. 18–31, March 1995.
[2] J. Rushby, "Formal methods and their role in the certification of critical systems," Safety and Reliability of Software Based Systems (Twelfth Annual CSR Workshop), Tech. Rep., 1995.

[3] D. Fensel, A. Schnegge, R. Groenboom, and B. Wielinga, "Specification and verification of knowledge-based systems," in *Proceedings of the 10^{th} Banff Knowledge Acquisition for Knowledge-Based System Workshop*, 1996, pp. 18–20.

[4] M. Fitting, *First-order logic and automated theorem proving (2nd ed.)*. Springer-Verlag New York, Inc., 1996.

[5] M. D. Rijke, "Handbook of tableau methods, marcello d'agostino, dov m. gabbay, reiner hähnle, and joachim posegga, eds." *J. of Logic, Lang. and Inf.*, vol. 10, pp. 518–523, September 2001.

[6] R. M. Smullyan, *Logical Labyrinths*. A K Peters, Ltd., Wellesley, MA, USA, 2009.

[7] J. Schumann and R. Letz, "Partheo: a high-performance parallel theorem prover," in *Proceedings of the tenth international conference on Automated deduction*, ser. CADE-10. New York, NY, USA: Springer-Verlag New York, Inc., 1990, pp. 40–56.

[8] P. Wolper, "Temporal logic can be more expressive," in *Proceedings of the 22nd Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1981, pp. 340–348.

[9] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *Proceedings of the 8th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '81. New York, NY, USA: ACM, 1981, pp. 164–176.

[10] E. A. Emerson and J. Y. Halpern, "Decision procedures and expressiveness in the temporal logic of branching time," in *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, ser. STOC '82. New York, NY, USA: ACM, 1982, pp. 169–180.

[11] V. Goranko, "Temporal logics for specification and verification," in *Proceedings of the European Summer School in Logic, Language and Information (ESSLI'09)*, 2009.

[12] P. Abate, R. Goré, and F. Widmann, "One-pass tableaux for computation tree logic," in *Proceedings of the 14th international conference on Logic for programming, artificial intelligence and reasoning*, ser. LPAR'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 32–46.

[13] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*. New York, NY, USA: Cambridge University Press, 2004.

[14] S. Schwendimann, "A new one-pass tableau calculus for pltl," in *Automated Reasoning with Analytic Tableaux and Related Methods*, ser. Lecture Notes in Computer Science, H. de Swart, Ed. Springer Berlin / Heidelberg, 1998, vol. 1397, pp. 277–291.

[15] V. Goranko, A. Kyrilov, and D. Shkatov, "Tableau tool for testing satisfiability in ltl: Implementation and experimental analysis," *Electron. Notes Theor. Comput. Sci.*, vol. 262, no. Elsevier Science Publishers B. V., pp. 113–125, May 2010.

[16] B. Beckert, R. Hähnle, P. Oel, and M. Sulzmann, "The tableau-based theorem prover 3tap version 4.0," in *Proceedings of the 13th International Conference on Automated Deduction: Automated Deduction*, ser. CADE-13. London, UK: Springer-Verlag, 1996, pp. 303–307.

[17] L. O. Astrachan and W. D. Loveland, "Meteors: High performance theorem provers using model elimination," Durham, NC, USA, Tech. Rep., 1991.

[18] S. Bose, E. Clark, D. Long, and S. Michaylov, "Parthenon, a parallel theorem prover for non-horn clauses," in *Proceedings of the Fourth Annual Symposium on Logic in computer science*. Piscataway, NJ, USA: IEEE Press, 1989, pp. 80–89.

[19] M. E. Stickel, "A prolog technology theorem prover: implementation by an extended prolog compiler," in *Proc. of the 8th international conference on Automated deduction*. New York, NY, USA: Springer-Verlag New York, Inc., 1986, pp. 573–587.

[20] R. Letz, G. Stenz, and A. Wolf, "P-setheo: Strategy parallel automated theorem proving," in *Proceedings of the International Conference on Automated Reasoning with Analytic Tableaux and Related Methods (TABLEAUX-98), Lecture Notes in Computer Science*, vol. 1397. Berlin-Heidelberg: Springer-Verlag, 1998, pp. 0–32.

[21] L. F. d. Cerro, D. Fauthoux, O. Gasquet, A. Herzig, D. Longin, and F. Massacci, "Lotrec: The generic tableau prover for modal and description logics," in *the Proceedings of the First International Joint Conference on Automated Reasoning*, ser. IJCAR '01. London, UK: Springer-Verlag, 2001, pp. 453–458.

[22] Z. Li, "Efficient and generic reasoning for modal logics," Ph.D. dissertation, University of Manchester, Faculty of Engineering and Physical Sciences, School of Computer Science, 2008.

[23] D. Tsarkov and I. Horrocks, "Fact++ description logic reasoner: System description," in *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*. Springer, 2006, pp. 292–297.

[24] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz, "Pellet: A practical owl-dl reasoner," *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 5, no. 2, pp. 51–53, 2007.

[25] P. Wolper, "The tableau method for temporal logic: an overview," *Logique et Analyse, 28(110–111)*, pp. 119–136, 1985.

[26] B. Carpenter, "mpijava home page," available at url: http://www.hpjava.org/mpiJava.html. Last accessed: June, 2011.

[27] J. Schumann, "Sicotheo - simple competitive parallel theorem provers based on setheo," Parallel Processing for Artificial Intelligence 3, Machine Intelligence and Pattern Recognition, Tech. Rep., 1995.

[28] M. Z. Islam, A. S. Mashiyat, K. N. Khan, and S. M. M. Karim, "Towards a tableau based high performance automated theorem prover," in *the 13th International Conference on Computer and Information Technology (ICCIT 2010)*, 2010, pp. 406–411.

Md Zahidul Islam is currently a M.Sc. student at StFX University, Antigonish, Canada. He received a B.Sc.Engg. degree in Computer Science and Engineering(CSE) from Khulna University, Bangladesh in 2007.

Currently he is working as a research assistant at CLI of StFX University, Canada. He is also a Lecturer at CSE Discipline of Khulna University, Bangladesh (now on study leave). After receiving his B.Sc.Engg. degree he has worked as a Part-time Lecturer at Khulna University and as a software engineer at Dohatec New Media, Bangladesh. His areas of interest include formal verification, software engineering, and data mining.

Ahmed Shah Mashiyat has just finished his Masters in Computer Science at St. Francis Xavier University (StFX). Before joining StFX he was working as a Senior Software Engineer in Jaxara IT Ltd. He has a B.Sc. in CSE from Khulna University, Bangladesh. Ahmed is a Student Advisory Committee (SAC) Council Member of MITACS. He served as a president of StFX Grad Student Society for session 2009-2010. His research interests include Software Engineering, Formal Verification, Workflow Systems, and Access Control.

Kashif Nizam Khan received his M.Sc in Security and Mobile Computing under the Erasmus Mundus-NordSecMob Program from NTNU, Norway and AALTO (TKK), Finland in 2010. He received his B.Sc Engineering degree in Computer Science and Engineering from Khulna University in 2007. He has worked as a research assistant in AALTO and as a Lecturer in the Department of CSE in Ahsanullah University of Science and Technology and Khulna University. His research interests include information security, mobility management, formal verification, next generation mobile technologies (3G, 4G) and future internet.

S. M. Masud Karim has been serving as a faculty member of Computer Science and Engineering (CSE) Discipline, Khulna University, Khulna, Bangladesh. He completed his B.Sc.Engg.(CSE) degree with distinction in 2001. He went abroad for higher studies in 2006 and was awarded M.Sc. in Media Informatics from RWTH Aachen University, Germany in 2008 and M.Sc. in Informatics from University of Edinburgh, UK in 2009. He is the author of international research papers published in international journals and conference proceedings. His areas of interest include information retrieval, data exchange, data integration, computer security.